CS 6961: Structured Prediction Spring 2017

Project Report

Siddartha Ravichandran (u1015163)
Sonam Choudhary(u1069221)

April 29, 2017

1 Image Captioning using Recurrent Neural Networks

The goal of our project is to study recurrent neural networks as a means to learning a model to enable captioning of images. By building this model, when queried with an image we hope to predict a corresponding caption or at the least score the image against 5 possible captions with the hope that the closest caption (or the one that actually fits the image) is scored highest. Recurrent neural networks (RNNs) are popular models that have shown great results in several natural language processing (NLP) tasks and we refer to several of its literature already applied to the task of image captioning.

2 Dataset

We use the Microsoft COCO (common objects in context) dataset to train our and validate our model. The dataset is further split into 2 sets, one each for training and validation. The training set contains about 83000 images and the validation set contains about 40000 images. Considering our task of image captioning, our labelled dataset consists of a mapping from each of the images to 5 of its corresponding valid captions.

For instance,

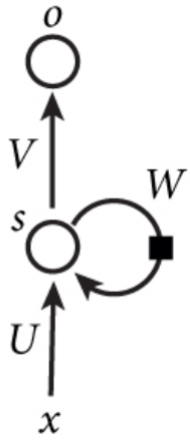
- A tennis player in front of other people on the side lines.
- A tennis player is wiping his racket off with a towel.
- A man in all white holding a racket.
- A good looking tennis player and his friends.
- Man holding a towel and a tennis racket, men in the back looking through backpacks and sitting on chairs.

3 Literature

Our main references for this task are 2 papers: *Deep Visual-Semantic Alignments for Generating Image Descriptions (Karpathy & Fie-Fie Li)* and *Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge (Vinyals et al).*

Both these papers have similar models, in that the first one uses an RNN and the second uses an LSTM (Long Short-term Memory) network, of which the second is stronger. We base our implementation on a simple RNN network which attempts to learn a language model based on the input captions with the image as the initial context.

This represents a 1-layer RNN, where $\boldsymbol{x}$ represents the input vectors corresponding to a particular element of the sequence, $\boldsymbol{o}$ represents the output probabilities (which is used to predict the next element of the sequence), $\boldsymbol{s}$ is a hidden state, and the vectors $\boldsymbol{U}$, $\boldsymbol{W}$ and $\boldsymbol{V}$ represent the current state of the neural network. The neural unit is unrolled to form a multi-layer system depending on the length of the sequence.



What is important to note that there can be numerous hidden states. The parameters learned by the RNN represent the weights that are fed to the hidden states along with the input vectors. These hidden states which can be given by any activation unit like tanh, sigmoid, etc. like generic neural networks, but the parameters remain constant across the sequence making it different from a generic neural network. The hidden state can be considered the memory of the system, that is, what the network has seen so far in the sequence.

4 Data Preprocessing

Since the dataset is large, we preprocess the data to extract the raw image data and also build our understanding of the vocabulary, along with an indexed mapping between the images and the captions. The mapping from images to captions is contained in a .json file, which along with the images (.jpg files) are then pre-processed to obtain another .json file and a .h5 file.

The output .json files contains information about the vocabulary or basically information extracted from the captions, like number of words, the total occurrences of the each word (words that occur very few times are ignored), etc.. The .json file also contains information which maps each word to an index and vice versa. It also contains an indexed mapping of

the images and its corresponding mapping with the captions.

The .h5 contains the raw image data which is contained as 3x256x256 sized arrays for each image, corresponding to the RGB values of the image resized to 256x256 pixels.

The output of the pre-processing is used as inputs to the training script.

5 How does the RNN help?

We shall cover some aspects of the RNN in the analysis section but, just briefly an RNN like we mentioned above, works well in learning a model (defined by parameters) that can weigh a sequence or in other words can be used to generate or sample a sequence by using the probabilities. Specifically, given a partial sequence at a particular time step, it can predict the next element in the sequence.

But we are given an image and its corresponding sequence during the training stage. So essentially we want to learn the sequence of the caption given the context of the image. Assuming we can map our feature representation of images and the individual words of the caption to a common space, we can feed into the RNN a sequence in the following order:

$$[\textbf{Image}] \rightarrow [\textbf{Caption}_0] \rightarrow [\textbf{Caption}_1] \rightarrow [\textbf{Caption}_2] \ldots \rightarrow [\textbf{Caption}_n]$$

where [ ] represents a common embedded space.

6 Model

Now that we have established how our RNN will be used, lets see the entire model. A common step that will be used both during training and prediction is that of converting our image to the common embedded space that we were just talking about earlier. To do so, we use a pre-trained model:a convoluted neural network (CNN), namely the VGG(visual graphics group at Stanford) Net CNN: *Very Deep Convolutional Networks for Large-Scale Image Recognition.*

While the aim of the VGG CNN is to provide a multi-class classifier across numerous object types (like dog, cat, table, etc.), we extract the vector representation of the image that the CNN produces.

6.1 Training our model

To train our model we feed the RNN with the sequence as follows:

$$[\textbf{Image}] \rightarrow [\textbf{Caption}_0] \rightarrow [\textbf{Caption}_1] \rightarrow [\textbf{Caption}_2] \ldots \rightarrow [\textbf{Caption}_n]$$

where [ ] represents a common embedded space.

For every image-caption pair that we feed-forward through the RNN, we perform back-propagation followed by updating the networks parameters using techniques like stochastic gradient descent(SGD) or *Adam*.

6.2 Using our model

With the trained RNN unit in hand, we feed the image into it to predict the 1^{st} word of the caption, followed by using **image—caption**$_0$ sub-sequence to predict the 2^{nd} word of the caption and so on.

7 Technology Used

The primary neural network package we use is **TORCH**, and the RNN package written on top of it provided by *Element-Research/rnn* (git-hub repository name).

We also use some of the utilities for data loading provided by *neuraltalk2*(git-hub repository name), which implements an improved version of the model released by the Google Brain team (CNN + LSTM: the 2^{nd} paper we reference in the literature section).

The scripts for training and using the model are predominantly written in LUA and python for data preprocessing. We use loadcaffe package to load and build the VGG CNN (using the 16 layer model provided at `http://www.robots.ox.ac.uk/~vgg/research/very_deep`).

We use the optim package within torch to perform the optimization/parameter update using the various techniques provided such as *Adam, SGD, etc.*.

8 Implementation Details and Numbers

We use a data loader to obtain the raw data into memory. We predominantly use tensors to contain our data and vectors needed to represent the various parameters, states of the RNN.

Information we have about the caption vocabulary:

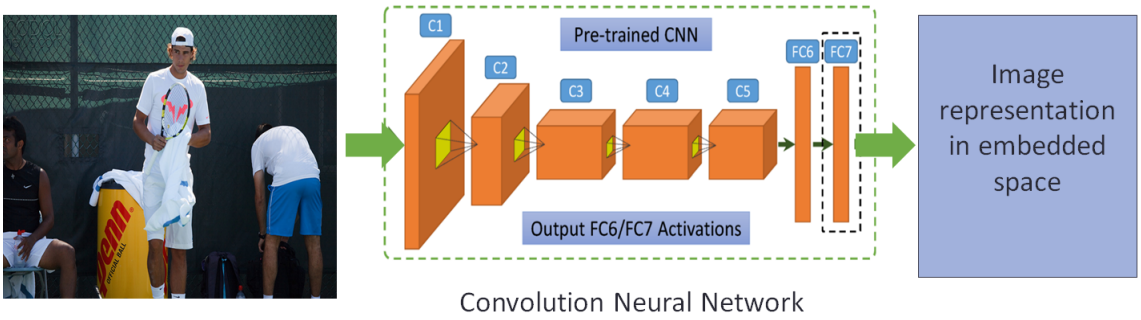
- Vocabulary size : 9567 (+1 for start token)
- Max. caption length : 16
- A mapping from index to word and word to index for each word in the vocab

We define our RNN with the following parameters:

- Number of hidden states : 512
- Output space : 9567 (+1) – probability of each word in the vocab

The inputs can be batched for efficiency, say of size N. N essentially represents the number of images being used at one time-step. Therefore there are N x 5 captions being used.

Our raw image data is given by a 4D-Tensor of size $N * 3 * 256 * 256$



8.1 Forwarding CNN and RNN

The output of the CNN for a given image is an array of size 512 elements, in our case of a 2D-Tensor of size $N * 512$, expanded to $5N * 512$, to create every image-caption pair.

Our captions are given by a 2D-tensor of size $16 * 5N$, each column representing a caption, wherein 16 is the max. caption length. 0 is used to fill the remaining space for shorter captions. The captions are represented by the index of the words (maintained in the vocab information) that make it.

In order to move the captions to a common space, we use a LookUpTable configured by 512 and the vocab size(9567). The output of the lookup is a 3D-tensor of size $16 * 5N * 512$ for every word of the $5N$ captions. We prepend this array with the image's representation making it $17 * 5N * 512$ (lets name this M).

M is then fed into our RNN, which on unrolling will form a 17-layer network, from which we compute the log-probabilities of what the RNN predicted the sequence to be. This is given by a 3D-tensor of size $17 * 5N * 9567$. Basically, this corresponds to the probability of each word being the next in the predicted sequence.We then compute the loss (how far the output is from the caption itself) by using a Criterion based on negative log likelihood. Find below a code snippet of how all this happens.

```
-- forward the ConvNet on images (most work happens here)
local image_features = cnn:forward(data.images)
-- we have to expand out image features, once for each sentence
local expanded_feats = expander:forward(image_features)

-- forward the model

-- add start tokens to the caption
local startTokens = torch.LongTensor(1,data.labels:size(2)):fill(loader:getVocabSize() + 1)
local labels = torch.cat(startTokens, data.labels:clone(), 1)

-- step to prevent lookup from crashing
labels:apply(function(x) if x == 0 then return 1 else return x end end)

-- a tabular representation of the caption
local curTargets = targetmodule:forward(labels)

-- move caption to common embedded space by forward through lookup table
local flabels = lookup:forward(data.labels:apply(function(x) if x == 0 then return 1 else return x end end))

-- create combined image-caption sequence
local a = torch.reshape(expanded_feats,torch.LongStorage{1,expanded_feats:size(1),expanded_feats:size(2)})
local b = torch.cat(a,flabels,1)

-- forward rnn with the above created sequence
local logprobs = lm:forward(b)

-- forward the model criterion
local loss = crit:forward(logprobs, curTargets)
```

8.2 Back-propagating RNN

As we would in regular neural networks, we back-propagate through the network, followed by updating the parameters of the network using a technique like SGD. Find below a code snippet of the same.

```
-- backprop criterion
local dlogprobs = crit:backward(logprobs, curTargets)

-- backprop model and lookup table
lookup:backward(labels,lm:backward(flabels, dlogprobs))

-- update the rnn parameters using sgd
sgd(params, grad_params, learning_rate)
```

8.3 Predicting caption

At prediction time, given an image we want to predict the sequence as seen below. We forward the image through the CNN as required followed by predicting the sequence.

```
-- forward RNN with image_features (CNN) to extract 1st word of sequence
local image_out = lm:forward(image_features)

local sampleLogProbs, it

-- find word with maximum probability
sampleLogProbs,it = torch.max(torch.cat(image_out,2),1)
it = it:view(-1):long()

-- set 1st word of sequence
seq[1] = it

-- iterate over sequence length to predict rest of the sequence
for t=2,loader:getSeqLength() do

    local out, next_word, wordLogProbs, wordLookup

    -- move sequence to common embedded space
    wordLookup = lookup:forward(seq)

    -- prepend sequence with image features
    local a_l = torch.reshape(feats,torch.LongStorage{1,feats:size(1),feats:size(2)})
    local b_l = torch.cat(a_l,wordLookup,1)

    -- forward model with image-caption sequence(whatever we have so far)
    out = lm:forward(b_l)

    -- get the output probabilities of the current timestep
    out = out[t]

    -- again find word with max. probability of being the (t+1)th entry in the sequence
    wordLogProbs, next_word = torch.max(out,2)
    next_word = next_word:view(-1):long()

    -- set next_word
    seq[t] = next_word

    -- continue until complete sequence found
end
```

9 Training a model

We run our scripts on a 8-core system with a Tesla K80 GPU (on Google cloud platform's compute engine). Our batch size is 32 images (or 160 captions) per iteration. We measure loss for each iteration and validate about 3200 (from the 40000) images randomly after every 1000 iterations.

10 Results

While our trained model does not yet predict captions well (many repetitive words), it scores them well. We got our model to train until loss was about 0.0001. Find below a scoring of images:

Caption	Score
Man riding a motor bike on a dirt road on the countryside.	6.12
Man is siting on a bike.	5.98
A man is walking on the road.	11.34
A boy is playing with a ball.	3.58
A man is sitting on a bench.	7.54

Caption	Score
A person holding a cell phone in their hand.	8.09
A man in wearing a black shirt.	9.82
A person is talking on phone.	14.28
A man is eating food.	10.36
A book is kept on the table.	4.65

11 Analysis

- We need to spend more time on cross-validation. Since the model takes up to 2 days to train, we were not able to cross-validate across parameters like learning rate. Several times, we either skipped the minima and our loss started to increase. The decay rate also needs to be validated.
- We used a simple RNN to build a model, but would like to explore and apply other concepts like LSTMs, gated recurrent units, multi-function recurrent unit.
- While we spent a lot of time in class with HMMs, an RNN conditions the generation of each word on the entire previous history of words generated. A Markov chain only conditions on a fixed window. Perhaps a particular RNN will learn to truncate its conditioning context and behave as a Markov chain, or perhaps not; but RNNs in general certainly can generate languages that Markov chains cannot. This is an interesting contrast and we would like to study this in terms of actual figures.
- We also thought of different ways of maintaining the context of the image. Basically currently we pass in the image features at the very beginning. Will passing it in at a different point or possibly at several intervals improve the model?
- Our dataset was very generic, possibly using a smaller and a more dedicated dataset would be better in terms of understanding the inner aspects of RNNs.